PCT/IB04/51244

MC 030937

Europäisches Patentamt

European Patent Office

Office européen des brevets

Bescheinigung

Certificate

Attestation

Die angehefteten Unterlagen stimmen mit der ursprünglich eingereichten Fassung der auf dem nächsten Blatt bezeichneten europäischen Patentanmeldung überein.

The attached documents are exact copies of the European patent application described on the following page, as originally filed.

Les documents fixés à cette attestation sont conformes à la version initialement déposée de la demande de brevet européen spécifiée à la page suivante.

Patentanmeldung Nr. Patent application No. Demande de brevet n°

03102240.3

PRIORITY DOCUMENT
SUBMITTED OR TRANSMITTED IN COMPLIANCE WITH RULE 17.1(A) OR (B)

REC'D 23 JUL 2004
WIPO PCT

Der Präsident des Europäischen Patentamts;
Im Auftrag

For the President of the European Patent Office

Le Président de l'Office européen des brevets
p.o.

[illegible]

R C van Dijk

Anmeldung Nr:
Application no.: 03102240.3
Demande no:

Anmeldetag:
Date of filing: 21.07.03
Date de dépôt:

Anmelder/Applicant(s)/Demandeur(s):

Koninklijke Philips Electronics N.V.
Groenewoudseweg 1
5621 BA Eindhoven
PAYS-BAS

Bezeichnung der Erfindung/Title of the invention/Titre de l'invention:
(Falls die Bezeichnung der Erfindung nicht angegeben ist, siehe Beschreibung.
If no title is shown please refer to the description.
Si aucun titre n'est indiqué se referer à la description.)

A method of searching in an collection of documents

In Anspruch genommene Prioriät(en) / Priority(ies) claimed /Priorité(s) revendiquée(s)
Staat/Tag/Aktenzeichen/State/Date/File no./Pays/Date/Numéro de dépôt:

Internationale Patentklassifikation/International Patent Classification/
Classification internationale des brevets:

G06F17/30

Am Anmeldetag benannte Vertragstaaten/Contracting states designated at date of filing/Etats contractants désignées lors du dépôt:

AT BE BG CH CY CZ DE DK EE ES FI FR GB GR HU IE IT LU MC NL
PT RO SE SI SK TR LI

A method of searching in a collection of documents

The invention relates to a method of searching in a collection of documents, the documents having a tree-like structure and each document in the collection of documents complying with at least one document structure definition in a collection of document structure definitions, and in particular to a method comprising the steps of receiving a certain branch and searching for at least part of the certain branch in documents.

The invention further relates to a computer program product enabling a programmable device to carry out a method of searching in a collection of documents.
The invention also relates to a method of indexing a collection of documents, an in particular to a method enabling a method of searching in a collection of documents.

The invention further relates to a computer program product enabling a programmable device to carry out a method of indexing a collection of documents.

An example of such a method of searching in a collection of documents is known from a World Wide Web Consortium standard called XPath. This standard describes searching for XML documents containing a certain path. XML documents have a tree-like structure, wherein each node has a tag and possibly a value. There is not more than one path between each two nodes. It is general practice to search for a path in XML documents by searching in each individual XML document. It is a drawback of the known method of searching in a collection of documents that a search may take a long time, especially if the documents are encrypted and need to be decrypted.

It is a first object of the invention to provide a method of searching in a collection of documents, which enables a more efficient search.

It is a second object of the invention to provide a method of indexing a collection of documents, which enables a more efficient search.

According to the invention, the first object is realized in that the method of searching in a collection of documents comprises the steps of: receiving a certain branch; determining a subset of the collection of document structure definitions, each document structure definition in the subset allowing the certain branch to exist in a document complying to the document structure definition; determining a subset of the collection of documents, the subset of documents comprising all documents of the collection of documents complying to any one of the document structure definitions in the subset; and searching for at least part of the certain branch in each document. While a path starts at one node and ends at another node, a branch may start or end at more than one node. Both a path and a branch comprise one or more tags and a path is also a branch. A path may, for example, be represented like 'book/name' or 'book.name'. A branch may, for example, be represented by 'book/name', 'book.name', or 'book(name+author(name+age))'. A branch may be represented by a plurality of paths, like, for example, {'book.name','book.author.name', book.author.age'}. A document may for example be an XML or an SGML document. A document structure definition may for example be an XML Data Type Definition (DTD) or an XML schema. By using the document structure definition to determine a set of candidate documents, a search is made more efficient. It is no longer necessary to search in all the documents themselves.

In an embodiment of the method of searching in a collection of documents of the invention, a further step comprises attempting to decrypt each encrypted document in the subset of documents. If documents in the collection of documents are encrypted, this step decrypts the candidate documents. Because not all encrypted documents have to be decrypted, unnecessary decryption of encrypted documents is reduced.

The step of determining a subset of the collection of documents may comprise calculating a number for at least part of the certain branch by applying a hash function to the at least part of the certain branch and looking up which documents are mapped to the calculated number in a mapping from number to documents, the mapping being associated with a document structure definition of the subset of document structure definitions and the documents in the mapping complying to the document structure definition. This not only provides security, the mapping does not show which branch is present in which document, but also allows efficient looking up of the documents. The size of the hash mapping may be adapted based on the collection of documents.

The method may comprise a further step receiving a certain value associated with the certain branch, the mapping may further comprise an association between a document in the mapping and a value domain partition, and the step of determining a subset of the collection of documents may further comprise checking whether a value domain partition associated to a document mapped to the calculated number matches a further value domain partition, the further value domain partition comprising the received value. Security is provided by not placing a value into the mapping, but only a value domain partition. The value domain partition only gives a weak indication of possible values, but does enable a more efficient search. A value domain partition may for example be 'a-e', 'a,b,c,d,e', '1-5', '1,2,3,4,5', 'Europe', or 'Netherlands, Germany, France, ...'.

The step of determining a subset of the collection of documents may comprise looking up, in a mapping from document structure definition to documents, which documents comply to any one of the subset of document structure definitions. This mapping may, for example, simply be a text document. Although not the most efficient solution, it can easily be created by hand.

The step of determining a subset of the collection of document structure definitions comprises calculating a further number for at least part of the certain branch by applying a further hash function to the at least part of the certain branch and looking up which document structure definitions are mapped to the calculated number in a mapping from number to document structure definitions. By using a second hash mapping, for example, in the form of a hash table, the step of determining a subset of the collection of document structure definitions is made more efficient. It is made unnecessary to decrypt document structure definitions. Security is still provided, as the second mapping does not show which branch is present in which document structure definition.

The step of determining a subset of the collection of document structure definitions may comprise attempting to decrypt each encrypted document structure definition in the collection of document structure definitions and attempting to determine for each document structure definition whether the document structure definition allows the certain branch to exist in a document complying to the document structure definition. If the amount of indexing should be limited, already existing XML DTD files may, for example, be used in a search. Each XML DTD file that has successfully been decrypted or has not been encrypted may, for example, be read and represented by a tree. In this tree, each path that is allowed by the XML DTD should be present at least once. The tree may be traversed to determine whether the XML DTD allows the certain branch to exist in an XML document complying to the XML DTD.

According to the invention, the second object is realized in that the method of indexing a collection of documents comprises the steps of: creating an empty index for each document structure definition of the collection of document structure definitions, the index mapping each integer from a range of integers to a document of the collection of documents; calculating a number for at least a part of a branch in a document of the collection of documents by applying a hash function to the at least part of the branch, the number being limited to the range of integers and the calculation possibly producing a same number for different branches; and creating an entry in an index for a document structure definition to which said document complies, the entry comprising a mapping from said calculated number to said document comprising the branch.

In an embodiment of the method of indexing a collection of documents of the invention, creating an entry in the index comprises associating the document in the mapping to a value domain partition, the value domain partition comprising a value associated with the branch.

The method may comprise a further step comprising creating an empty further index in which each integer from a further range of integers can be mapped to a document structure definition; a further step comprising calculating a further number for at least part of said branch by applying a further hash function to said branch, the further number being limited to the further range of integers and the calculation possibly producing a same further number for different branches, and a further step comprising creating an entry in the further index, the entry in the further index comprising a mapping from the calculated further number to said document structure definition to which said document complies.

By using the document structure definitions in the indexing process, the indexing of the collection of documents is performed efficiently. In an alternative method of indexing a collection of documents, one index mapping from branch to documents may be created instead of two indices. This one index might, for example, be a hash table, thereby providing security and efficiency. The alternative range of integers used in such a table will most likely be larger than said range of integers or said further range integers. Instead of using a hash table, another way of mapping could be used. An alternative index might for example comprise a mapping from branch (name) to documents. This might also enable a more efficient search in comparison with a search without using an index, but it seems less advantageous than using a hash table. To provide security, extra measures need to be taken. To ensure that a branch does not unambiguously map to a document, a branch may have to be mapped to documents that do not contain the branch.

1 Introduction

Building trust and confidence is a key aspect for the development of large-scale reliable Web-based applications like Semantic Web. Alongside, XML is becoming the dominant standard for describing and interchanging data between various systems and databases on the Internet [3]. The sheer volume of such XML-formatted data available necessitates the development of XML security techniques to protect them from being disclosed and tampered with. One prototypical technique for building security and safety is to distribute and store these XML-formatted data in encrypted form [10]. W3C recommends an ``XML Encryption Syntax" to allow the encryption of XML data using a combination of symmetric and public keys, where element content is encrypted by means of a symmetric key that in turn is encrypted by means of the public key of the recipient [4, 5]. Nevertheless, securing XML data in cipher text should not hinder its efficient processing for various applications. The deployed security techniques should on the one hand satisfy the security requirements on XML data, while at the same time allow efficient manipulation of the data without loss of confidentiality.

Since query is a fundamental operation that is carried out on XML data, a first step to proceed is to address the issue around effective and efficient querying of encrypted XML data. A straightforward approach to search on encrypted XML data is to decrypt the cipher text first, and then do the search on the clear decrypted XML data. However, this inevitably incurs a lot of unnecessary decryption efforts, leading to a very poor query performance, especially when the searched data is huge, while the search target comes only from a small portion of it. To solve this problem, we raise two questions here: (1) "*Can we discard some non-candidate XML data and decrypt the remaining data set instead of the whole data set?*" (2) "*If so, how can we effectively and efficiently distinguish candidate XML data from non-candidate data?*"

There is some previous work in different research areas that are related to our work. [11] presents techniques to support *keyword*-based search on encrypted textual string. Recently, [8,9] explore techniques to execute *SQL*-based queries over encrypted relational tables in a database-service provider model, where an algebraic framework is described for query rewriting over encrypted attributed representation. However, compared to the problem to be addressed in this study, the functionalities provided by the above work are still very limited and insufficient in performing complex *XPath*-based XML queries over encrypted semi-structured XML data.

In this paper, we propose to augment encrypted XML data with *encodings* which characterize the topology and content of every encrypted tree-structured XML data. A hash-based approach is employed to compute hashed XML path information into encoding schemas. The filtering of non-candidate data set can then be performed by examining query conditions, expressed in terms of XPath expressions [6], against the encodings. We outline a generic framework for conducting efficient queries on encrypted XML data, which is comprised of three phases, namely, *query preparation*, *query pre-processing*, and *query execution*. The *query preparation* phase aims to prepare for efficient query answering by encoding XML DTDs and XML documents before they are encrypted and stored in the database. This phase runs off-line. When a query is issued at runtime, the *query pre-processing* phase filters out impossible query candidates, so that the decryption and query execution that the *query execution* phase undertakes can be more focused on potentially target XML data. We analyze the time and space complexity of the proposed query strategy, and demonstrate its effectiveness through a set of examples on both synthetic and real-life data.

The contributions of the paper are two folds. Developing an effective approach to enable efficient querying of encrypted XML data is only the tangible contribution of our study. The more important contribution of the paper is that, we bring together the worlds of XML data management and security, where the major challenge, we believe, lies in the development of techniques that deal with the interaction between the security and large-scale XML data management techniques in such a way that the seemingly contradictory requirements between security and efficiency are united [10]. Currently, both security and efficient processing of semi-structured XML data have gained increasingly in importance in building trusted Web-based applications, and combining the manifold areas points to a new direction with a wide spectrum of research questions. It is our fond hope that the work reported here could stimulate the interests in this area in the solid belief that a lot of relevant research can and needs to be done [10].

The remainder of the paper is organized as follows. Section 2 outlines the framework for efficient querying over encrypted XML data, which consists of three phases: i) query preparation, ii) query pre-processing, and iii) query execution. Section 3 and Section 4 describe in detail the query preparation phase and query pre-processing phase, respectively. Section 5 reports our performance study on both synthetic and real-life data. Section 6 concludes the paper with a brief discussion of future work.

2 A Framework for Querying Over Encrypted XML Data

In this section, we first present a high-level framework for efficient querying over encrypted XML data, and then outline the general philosophies that influence the design of each component in such a framework. We assume the availability of DTD for each XML document in this study. For security reason, all XML DTDs and XML documents are encrypted and stored in a database.

The Framework

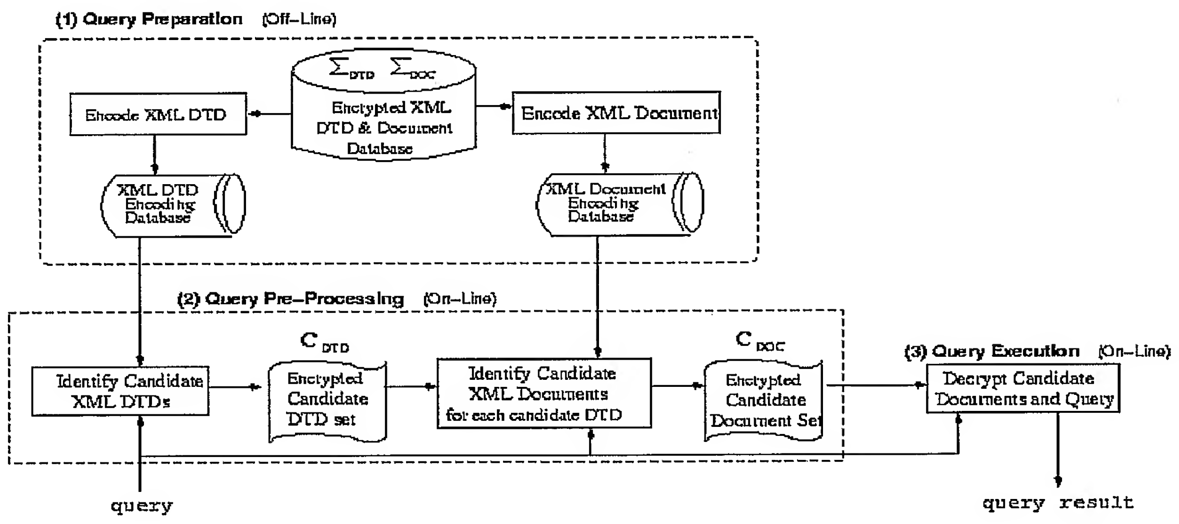

(1) Query Preparation (Off-Line)
Σ_{DTD} Σ_{DOC}
Encrypted XML DTD & Document Database
Encode XML DTD
Encode XML Document
XML DTD Encoding Database
XML Document Encoding Database
(2) Query Pre-Processing (On-Line)
C_{DTD}
C_{DOC}
Identify Candidate XML DTDs
Encrypted Candidate DTD set
Identify Candidate XML Documents for each candidate DTD
Encrypted Candidate Document Set
(3) Query Execution (On-Line)
Decrypt Candidate Documents and Query
query
query result

Figure 1. A framework for querying over encrypted XML data

Figure 1 shows a generic framework for conducting efficient queries on encrypted XML data. It is comprised of three phases: *query preparation, query pre-processing and query execution.*

- **Phase-1 (query preparation)**
  The aim of this phase is to prepare for efficient querying over encrypted XML data by encoding each XML DTD and associated documents before they are encrypted and stored in the database. Such an encoding is carried out in two steps, "*Encode XML DTD*" and "*Encode XML Document*". The coding results of XML DTDs and documents are stored in two separate databases, called "*XML DTD Encoding Database*" *and* "*XML Document Encoding Database*".
  In response to a query at run-time, these encodings can be used to pre-select potentially target documents without the need to decrypt the whole document set in the database. We call these potentially target documents *candidate documents* in the paper. Let Σ_{DTD} and Σ_{DOC} denote a finite set of encrypted XML DTDs and XML documents in the database. Detailed encoding schemas and their computation will be described in Section 3.

- **Phase-2 (query pre-processing)**
  It is obvious that decrypting all encrypted XML documents to answer a query inevitably incurs an excessive overhead, especially when the target data occupies only a small portion of the database. In order to make encrypted XML query processing truly practical and computationally tractable, and meanwhile preserve security, for each query, we incorporate a pre-processing stage, whose aim is to filter out impossible candidates so that decryption and query execution can be more focused on potentially target documents. Two steps are conducted in this phase. First, a set of candidate XML DTDs are identified through the step "*Identify Candidate XML DTDs*", which examines query conditions, expressed in terms of XPath expressions [6,7], against DTD encodings in the "*DTD Encoding Database*". Then, corresponding to each selected candidate DTD, the "*Identify Candidate XML Documents*" step further filters out candidate documents based on documents' encodings stored in the "*Document Encoding Database*". The candidate DTD set and document set returned are subsets of the original encrypted DTD set and document set, respectively. We use C_{DTD} and C_{DOC} to denote the set of candidate XML DTDs and documents for a query, where $C_{DTD} \subseteq \Sigma_{DTD}$ and $C_{DOC} \subseteq \Sigma_{DOC}$.
  Detailed descriptions of these two steps will be given in Section 4.

- **Phase-3 (query execution)**

The identified candidate DTDs and documents, returned from Phase-2, are decrypted into clear DTDs and documents, on which the query can be executed. We use T_{DTD} and T_{DOC} to denote the set of target DTDs and documents, from which the user's desired query results come.

In this phase, conventional XML query engines can be employed. Since querying over non-encrypted XML data has been widely investigated in the literature, we focus on Phase-1 and Phase-2 in this study.

2.2 Design Philosophies

In general, we have the following philosophical considerations that guide the design of Phase-1 and Phase-2.

Phase-1: Query Preparation (off-line)

- **Effective encoding schemas.** The encoding schemas devised in this phase should be effective enough to facilitate fast identification of candidate DTDs and documents given a query issued at run-time.
- **Safe encodings.** It must be guaranteed that encodings do not leak any information to externals. In other words, from the encodings, readers cannot learn anything about the encoded DTDs and XML documents.
- **Condensed encoding size.** The space used to store the encodings of XML DTDs and documents should be as small as possible.

Phase-2: Query Pre-Processing (on-line)

- **Completeness.** To ensure the correctness of query execution, the query pre-processing phase must be complete, which means that its pre-selected candidate DTDs and documents must constitute the supersets of the real target DTDs and documents for a given query.
- **High selectivity.** It is clear that for efficiency, Phase-2 should only deliver candidate DTDs and documents with high likelihood of being real target DTDs and documents, because for each selected candidate XML data, we need to decrypt it first and then query it.
- **Efficiency.** Efficient strategies are needed in this phase to speed up the candidate pre-selection process by checking the queries against the encodings of XML data.
- **Hidden query support.** For deep security, it is sometimes also desirable to realize a query over encrypted XML data even without revealing the query itself. That is, the query itself is encrypted. We leave this issue to our further work.

3 Query Preparation Phase

In this section, we propose a hash-based strategy to encode encrypted XML data during the query preparation phase. Based on the encodings obtained, the query pre-processing phase, which will be addressed in Section 4, can then effectively filter out query candidates, i.e., potential targets, from among a large set of blind documents in the database. Due to different characteristics and functions of XML DTDs and documents, we encode XML DTDs and XML documents separately using different encoding schemas.

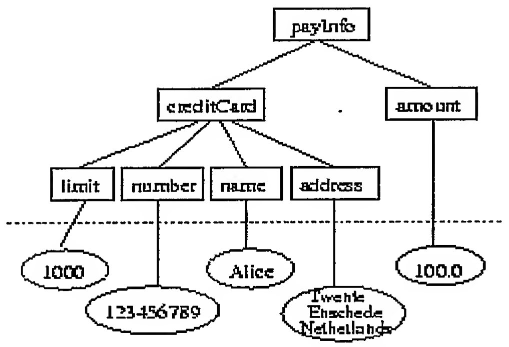
In the following, we first describe the computation method for encoding XML DTDs, followed by the method for encoding XML documents. For ease of explanation, a running example shown in Figure 2 is used throughout the discussion. A graphical representation of the DOM tree structure of the example DTD, dtd_1, together with its conforming document example doc_1 is shown in Figure 2(c).

```
<!DOCTYPE payInfo [
   <!ELEMENT payInfo (creditCard?, amount+)>
   <!ELEMENT creditCard (number, name, address)>
   <!ATTLIST creditCard limit CDATA #IMPLIED>
   <!ELEMENT number (#PCDATA)>
   <!ELEMENT name (#PCDATA)>
   <!ELEMENT address (#PCDATA)>
   <!ELEMENT amount (#PCDATA)>
]>
```

(a) An XML DTD exmple – dtd1

```
<payInfo>
  <creditCard limit=1000>
       <number> 123456789 </number>
       <name> Alice </name>
       <address> Twente 7500 AE, Netherlands </address>
   </creditCard>
  <amount> 100.0 </amount>
</payInfo>
```

(b) An XML document example – doc1 that conforms to dtd1



(c) A graphical representation of the DOM tree structure for dtd1 and doc1

Figure 2. A running example of an XML document with its DTD

3.1 Encoding XML DTDs

An XML DTD defines the legal building blocks of its conforming XML documents, like what elements, attributes, etc. are permitted in the documents [3]. These components construct a hierarchical tree structure that underlies the contents of the documents, with each path of the tree addressing a certain part of a document.
As the query pre-processing phase works on the basis of XPath expressions embedded in a query like XQuery, to prepare for efficient candidate selection, we thus take the strategy to encode each XML DTD in the unit of *path*. The notions of *path* and *path length* are defined as follows.

Definition 1. *A **path** p is a sequence of nodes $n_1, n_2,, \ldots, n_k$, denoted as $p = (n_1/n_2/ \ldots /n_k)$, where for any two consecutive nodes, n_i and n_{i+1} $(1 \leq i \leq k\text{-}1, k \geq 1)$, there exists an edge between them.*
*The **length** of path p, denoted as $|p|$, is the total number of edges in the path. That is, $|p = (n_1/n_2/ \ldots /n_k)| = k\text{-}1$.*

Table 1 lists the paths of various lengths, extracted from the example DTD *dtd_1* in Figure 2. Here, the content nodes under the dotted line are exempt from consideration, since they do not appear in *dtd_1*.

Path Length	Path
2	P_1=*(payInfo/creditCard/limit)* P_2=*(payInfo/creditCard/number)* P_3=*(payInfo/creditCard/name)* P_4=*(payInfo/creditCard/address)*
1	P_5=*(payInfo/creditCard)* P_6=*(payInfo/amount)* P_7=*(creditCard/limit)* P_8=*(creditCard/number)* P_9=*(creditCard/name)* P_{10}=*(creditCard/address)*
0	P_{11}=*(payInfo)* P_{12}=*(creditCard)* P_{13}=*(amount)* P_{14}=*(limit)* P_{15}=*(number)* P_{16}=*(name)* P_{17}=*(address)*

Table 1. Paths extracted from dtd_1

In essence, we use the technique of hashing on each path of an XML DTD to compose DTD encodings. Paths of different lengths will be hashed into different hash tables named $DTDHashTable_0$, $DTDHashTable_1$, $DTDHashTable_2$,, ..., $DTDHashTable_{max\_pathLen}$, respectively. All paths of length l (where $1 \leq l \leq max\_pathLen$), no matter which DTD it comes from, will share one single hash table $DTDHashTable_l$, with each bucket indicating a set of DTDs, whose paths have been hashed into the bucket. Suppose we have a path p extracted from dtd_1, the hash function $HashFunc(p)$ computes its hash value, i.e., bucket address in the hash table $DTDHashTable_{|p|}$. (Detailed computation of hash values will be given shortly.) Underneath the corresponding bucket, we link the identifier of dtd_1, signifying the DTD where p locates.

To filter out non-candidate DTDs for a query, we compute the hash values for all XPaths in the query using the same hash function, and then check the corresponding buckets in the DTD hash tables to obtain a subset of DTDs that possibly contain the requested paths. These DTDs are candidate DTDs to be considered for the query.

Algorithm 1 Hash function *HashFunc*(p)

Input: path $p = (n_1/n_2/ \dots /n_k)$, a fixed size s for node names, hash table size $SizeDTDHashTable_{|p|}$;

Output: hash value of p

```
1 For each node $n_i$ ($1 \le i \le k$), chop its name uniformly into an $s$-letter string
  $ChopName(n_i, s) = x_{n\,i,1}\, x_{n\,i,2} \dots x_{n\,i,s}$
  where $x_{n\,i,1}, x_{n\,i,2}, \dots, x_{n\,i,s}$ are letters in the name string of node $n$.

2 For each $s$-letter node name $x_{n\,i,1}\, x_{n\,i,2} \dots x_{n\,i,s}$, convert it into a decimal integer
  $Base26ValueOf(x_{n\,i,1}\, x_{n\,i,2} \dots x_{n\,i,s}) = offset(x_{n\,i,1})*26^{s-1} + offset(x_{n\,i,2})*26^{s-2} + \dots + offset(x_{n\,i,s})*26^0 = V_{n\,i}$,
  where $offset(x_{n\,i,j})$ ($1 \le j \le s$) returns the position of letter $x_{n\,i,j}$ among 26 letters.

3 Compute hash value of $p = (n_1/n_2/ \dots /n_k)$
  $HashFunc(n_1/n_2/ \dots /n_k) = (V_{n1} * 10^{k-1} + V_{n2} * 10^{k-2} + \dots + V_{nk} * 10^0) \bmod SizeDTDHashTable_{|p|}$
```

Algorithm 3.1 elaborates the procedures in computing the hash value for path $p = (n_1/n_2/ \dots /n_k)$. It proceeds in the following three steps.

First, node names in path p which could be of different lengths are uniformly chopped into the same size s, given by users as an input parameter, through the function *ChopName* (Algorithm 3.1, line 1). For example, let s=4, *ChopName*("*creditCard*", 4) = "*cred*", *ChopName*("*payInfo*", 4) = "*payI*", *ChopName*("*name*", 4) = "*name*".

Second, the chopped node name strings, which are of a fixed size after Step 1, are further converted into decimal integers via function *Base26ValueOf* (Algorithm 3.1, line 2). Example 1 explicates how it works when the size of node name string is set to 4.

Example 1 *When we let a 4-letter node name $x_1x_2x_3x_4$, which are case insensitive, represent a base-26 integer, we let the letter 'a' represent the digit-value 0, the letter 'b' represent the digit-value 1, the letter 'c' represent the digit-value 2, the letter 'd' represent the digit-value 3, and so on, up until the letter 'z', which represents the digit-value 25. Given a letter, function "offset" returns such a digit-value. The 4-letter node name $x_1x_2x_3x_4$ can thus be converted into a decimal integer using the formula:*

$$\text{Base26ValueOf}(x_1x_2x_3x_4) = \text{offset}(x_1)*26^3+\text{offset}(x_2)*26^2+\text{offset}(x_3)*26^1+\text{offset}(x_4)*26^0$$

Assume that $x_1x_2x_3x_4$ = "name", since the digit-values of 'n', 'a', 'm' and 'e' are offset('n') = 13, offset('a') = 0, offset('m') = 12, and offset('e') = 4 respectively, we have Base26ValueOf ("name") = $13*26^3 +0*26^2+12*26^1 +4*26^0 = 13*17576 + 0 + 312 + 4 =$ 228802.

In a similar way, we have Base26ValueOf ("cred") = $2*26^3+17*26^2+4*26^1+3*26^0 =$ $2*17576 + 17*676 + 104 + 3 = 35152 + 11492 + 104 + 3 = 46751$.

A general calculation of *Base26ValueOf* is:

$$Base26ValueOf(x_1\, x_2 \ldots x_s) = offset(x_1)*26^{s-1} + offset(x_2)*26^{s-2} + \ldots + offset(x_s)*26^0$$

Finally, hash function *HashFunc* derives the hash value of path $p = (n_1/n_2/ \ldots /n_k)$ based on the value V_{ni} returning from function *Base26ValueOf* on each node n_i (Algorithm 3.1, line 3).

$$HashFunc(n_1/n_2/ \ldots /n_k) =$$
$$(V_{n1} * 10^{k-1}+V_{n2} * 10^{k-2}+ \ldots +V_{nk} * 10^0) \bmod SizeDTDHashTable_{k-1}$$

To illustrate, let's see the following example.

Example 2 *Given a path p=(creditCard/name) where k=2 and $|p|=1$, let s=4 and $SizeDTDHashTable_{|p|} = SizeDTDHashTable_1=8$.*

Step 1: ChopName("creditCard", 4) = "cred", ChopName("name", 4) = "name".

Step 2: $Base26ValueOf(\text{“name”}) = 228802$, $Base26ValueOf(\text{“cred”}) = 46751$.

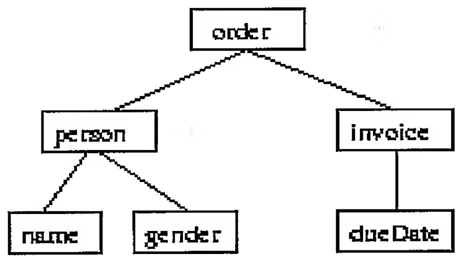
Step 3: $HashFunc(creditCard/name)$

$$= (Base26ValueOf(\text{“cred”})*10^1 + Base26ValueOf(\text{“name”})*10^{\wedge 0}) \bmod SizeDTDHashTable_1$$

$$= (46751*10 + 228802) \bmod 8 = 0$$

Therefore, path p=(creditCard/name) is hashed to the first bucket of the hash table $SizeDTDHashTable_1$. We link the identifier of dtd_1 under this bucket to indicate that dtd_1 contains a path whose hash value is 0.

```
<!DOCTYPE payInfo [
   <!ELEMENT order(person, invoice)>
   <!ELEMENT person(name, gender)>
   <!ELEMENT name(#PCDATA)>
   <!ELEMENT gender(#PCDATA)>
   <!ELEMENT invoice(dueDate)>
   <!ELEMENT dueDate(#PCDATA)>
]>
```



(a) Another DTD example – dtd2 **(b) A tree structure of dtd2**

Figure 3. Another DTD example dtd_2 with its DOM tree-structure

In order to provide a more complete overview on the hash-based encoding method, we introduce another DTD example dtd_2 shown in Figure 3. Using the same hash function, we hash all the paths from dtd_1 and dtd_2 with their residing DTDs marked in the respective buckets, as illustrated in Figure 4.

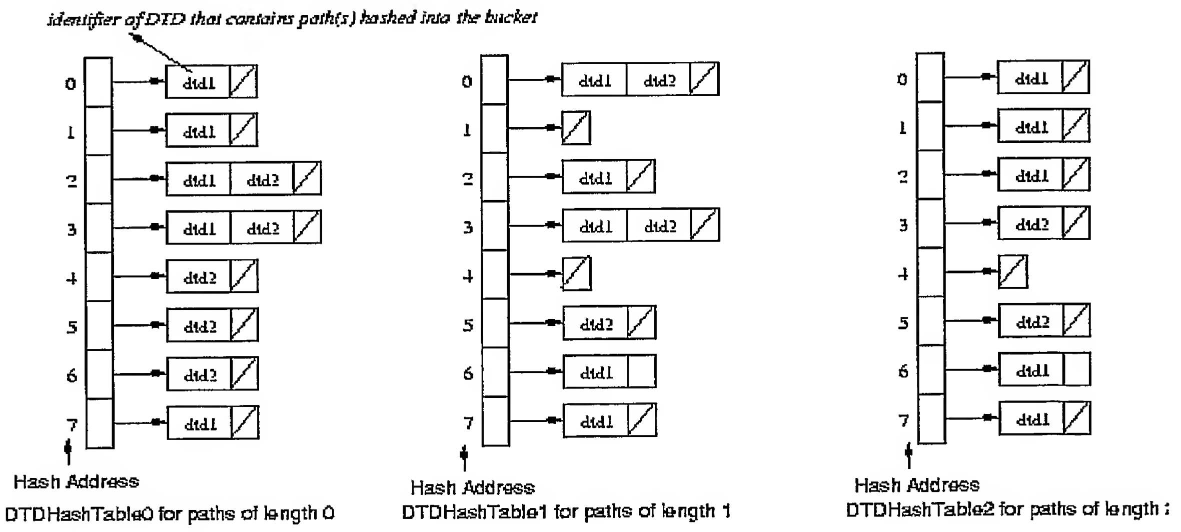

identifier of DTD that contains path(s) hashed into the bucket
0 1 2 3 4 5 6 7
dtd1
dtd2
Hash Address
DTDHashTable0 for paths of length 0
DTDHashTable1 for paths of length 1
DTDHashTable2 for paths of length 2

Figure 4. Encodings of the example dtd_1 and dtd_2 ($DTDHashTable_0$, $DTDHashTable_1$ and $DTDHashTable_2$)

3.2 Encoding XML Documents

XML documents that conform to one XML DTD possess a similar structure, but with possibly different element contents and/or attribute values to distinguish different documents. For instance, the conforming document doc_1 of dtd_1 shown in Figure 2 has a *limit* attribute of value 1000, represented as *limit*=1000 for simplicity. Its elements ***number***, ***name***, ***address*** and ***amount*** have contents 123456789, "*Alice*", "*Twente, Enschede, Netherlands*" and 100.0, respectively.

(Element/Attribute c_{name}, Content/Value c_{val})	$HashFunc(c_{name})$	$MapFunc(c_{val})$
c_1 = (*limit*, 1000)	0	1
C_2 = (*number*, 123456789)	1	10
C_3 = (*name*, "*Alice*")	0	0
C_4 = (*address*, "*Twente, Enschede, Netherlands*")	2	25
C_5 = (*amount*, 100.0)	1	7

Table 2. Pairs of element/attribute and content/value in doc_1, together with their hash and mapped values

After encoding XML DTDs, i.e., all possible paths each containing a sequence of nodes corresponding to elements or attributes, the second task of the query preparation phase is to encode their conforming documents, i.e., all pairs of element and element content (*element, element content*), attribute and attribute value (*attribute, attribute value*). Due to the different nature of contents, encoding documents is conducted in a different way from encoding DTDs, with the result stored in the "*Document Encoding Database*". We build a document hash table $DOCHashTable_{dtdi}$ for each dtd_i ($dtd_i \in \Sigma_{DTD}$). In the following, we describe the method of encoding a pair from dtd_1, $c = (c_{name}, c_{val})$ (where c_{name} denotes an element/attribute, and c_{val} denotes the corresponding element content/attribute value), into the hash table $DOCHashTable_{dtd1}$.

The hash address of each pair is calculated via function *HashFunc*(*p*) (Algorithm 3.1), using a different hash table size, which is $SizeDOCHashTable_{dtd1}$ rather than $SizeDTDHashTable_{|p|}$. In this case, path *p* always contains only one node, which is $p=(c_{name})$ and $|p|=0$. For example, let $s=4$, and the size of hash table $SizeDOCHashTable_{dtd1}$ equal to 4 (i.e., $SizeDOCHashTable_{dtd1}=4$). We have *ChopName*("*limit*")="*limi*". *Base26ValueOf*("*limi*") = $11*26^3 + 8*26^2 + 12*26 + 8 = 199064$, *HashFunc*(*limit*) = $199064*10^0 \bmod 4 = 0$.

After the derivation of bucket address in the hash table $DOCHashTable_{dtd1}$ from c_{name}, the entry to be put into the corresponding bucket is computed based on c_{val}, using the technique developed in [8]. The basic idea is to first divide the domain of node c_{name} into a set of *complete* and *disjoint* partitions. That is, these partitions taken together cover the whole domain; and any two partitions do not overlap. Each partition is assigned a *unique* integer identifier. The value c_{val} of element/attribute node c_{name} is then mapped to an integer, corresponding to the partition where it falls [8]. For example, we can partition the domain of attribute *limit* into [0, 500], (500, 1000], (1000, ∞) of identifier 0, 1, 2, respectively. The *limit* value 1000 is thus mapped to integer 1, and stored in the first bucket of $DOCHashTable_{dtd1}$, since *HashFunc*(*limit*)=0. The hash values for other pairs in the example document are calculated in the same way, which are shown in Table 2.

Note that the partition of a domain can be done based on the *semantics* of data and relevant applications. For instance, we can categorize the domain of element *name* according to the alphabetical order. The domain of element *address* can be partitioned according to *province* or *country* where located. In the current study, we enforce *order preserving* constraint on such a mapping "*MapFunc*: domain (c_{name}) → Integer", which means that for any two values c_{val1} and c_{val2} in the domain of c_{name}, if ($c_{val1} \leq c_{val2}$), then MapFunc(c_{val1}) ≤ MapFunc(c_{val2}).

Assume the mapping functions for *number*, *name*, *address* and *amount* return identifiers, as indicated in Table 2. Figure 5 plots the resulting encoding, i.e., $DOCHashTable_{dtd1}$, for the example XML document doc_1.

All documents that conform to one DTD share the same document hash table. The collision pairs are linked together underneath the bucket at the collision hash address.

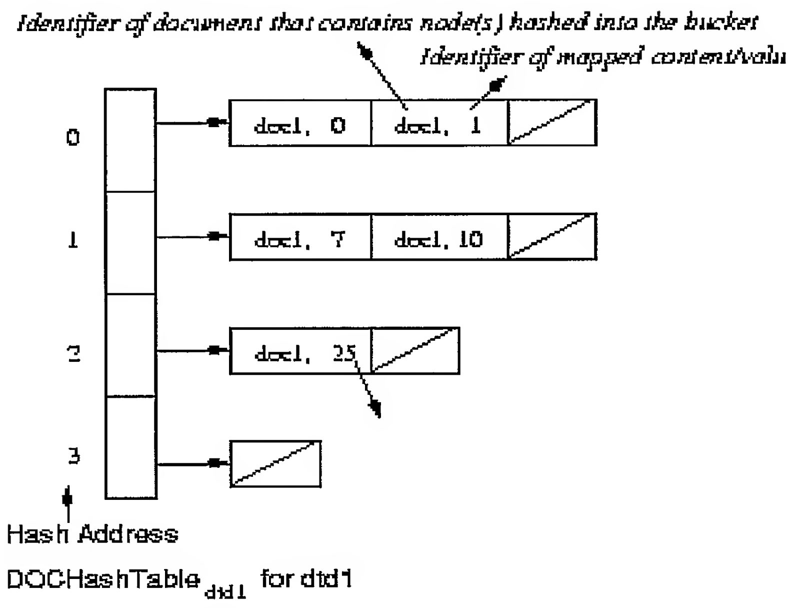

Identifier of document that contains node(s) hashed into the bucket
Identifier of mapped content/valu
0
1
2
3
doc1, 0
doc1, 1
doc1, 7
doc1, 10
doc1, 25
Hash Address
DOCHashTable dtd1 for dtd1

Figure 5. Encodings of the example document doc$_1$ from dtd$_1$ (*DOCHashTable$_{dtd1}$*)

4 Query Pre-Processing Phase

The aim of the query pre-processing phase is to identify candidate DTDs and documents by checking the query against the encodings of DTDs and documents, obtained after the query preparation phase. In this section, we first provide a brief description of XPath expressions used in query representation. We then discuss a method to match such XPath expressions to paths as described in Section 3 in order to facilitate candidate DTD and document selection. A two-step procedure is finally illustrated to identify candidate DTDs, followed by candidate documents for each selected candidate DTD.

XPath Expressions

The XPath language is a W3C proposed standard for addressing parts of an XML document [6]. It treats XML documents as a tree of nodes corresponding to elements/attributes, and offers an expressive way to specify and locate nodes within this tree.

XPath expressions state structural patterns that can be matched to paths, consisting of a sequence of nodes in the XML data tree [1, 2]. Such paths can be either absolute paths from the root of the data tree, or relative one starting with some known context nodes. The hierarchical relationships between the nodes are specified in XPath expressions using parent-child operator ("/") and ancestor-descendant operator ("//"). For example, the XPath expression *"/payInfo/creditCard/@limit"* addresses *limit* attribute of *creditCard* which is a child element of the *payInfo* root element in the document. The *name* element in the relative path expression *"//creditCard/name"* is a child relative to its parent *creditCard* element. The expression *"/payInfo//name"* addresses *name* descendant element of the *payInfo* root element.

XPath also allows the use of a wildcard operator ("*" or "@*"), which can match any element or attribute node with respect to the context node in the document data tree. In addition, predicates, enclosed in square brackets ("[ ]"), can be applied to further refine the selected set of nodes in XPath expressions. For example, *"/payInfo/creditCard[@limit<1000]/name"* selects the *name* element of the XML document if the attribute *limit* of *creditCard* has a value less than 1000.

Operators like ("|") and ("and") can also be applied to select constituent nodes of paths [6]. For instance, *"/payInfo/(creditCard|cash)/name"* expression selects every *name* element that has a parent that is either a *creditCard* or a *cash* element, that in turn is a child of a root element *payInfo*. On the contrary, *"/payInfo/creditCard[@limit and @dueDate]"* indicates all the *creditCard* children of the root element *payInfo* must have both a *limit* attribute and a *dueDate* attribute.

4.2 Mapping XPath Expressions to Paths

Considering that DTD encodings are computed against *paths* as defined in Definition 1 in Section 3, for efficient encoding-based query candidate pre-selection, we first need to match an *XPath expression* e, which is used to locate parts of a data tree, to a set of *paths* through the following three steps.

Step 1. Decompose XPath expression *e* into several ones at the point of "//" operator.
Since paths to be encoded during the off-line query preparation phase have only parent-child relationships ("/") between consecutive nodes (as shown in Table 1), we break an XPath expression from the points where the "//" operator locates, into several ones where each node, except for the first one, is prefixed only by "/". The resulting XPath expressions thus contain no ancestor-descendant relationships ("//") between every two consecutive nodes.

Example 3 *An XPath expression e="/payInfo[amount>100]//name" can be decomposed into two shorter XPath expressions e_1' = "/payInfo[amount>100]" and e_2' = "//name". We use e $\Rightarrow_1 e_1' \wedge e_2'$ to denote such a semantically equivalent decomposition.*

For ease of explanation, we signify the XPath expressions derived after Step 1 using a prime symbol like *e'*. They form the input of Step 2.

Step 2. Simplify predicate constraints in each XPath expression *e'* to only hierarchical relationships.

As DTD encoding relieves value constraints on path nodes, and focuses only on their hierarchical relationships, to facilitate candidate DTD filtering based on path encodings, we relax value constraints on nodes like *"[amount>100]"* and *"[@limit=1000]"*, specified in XPath predicate conditions, and keep only their inherent parent-child or element-attribute relationships.

Example 4 *The predicate constraint in e_1' = "/payInfo[amount>100]" implies that amount is a child element of payInfo, whose value constraint is eliminated by augmenting a parent-child relationship between payInfo and amount, resulting in a more relaxed XPath expression e_1'' = "/payInfo/amount" after Step 2. We use $\Rightarrow_2$ to denote such a simplification transformation, i.e., $e_1' \Rightarrow_2 e_1''$.*

Example 5 *A predicate situated in the intermediate of an XPath expression like "/payInfo[amount>100]/creditCard" leads to two XPath expressions being generated after Step 2, which are "/payInfo/amount" and "/payInfo/creditCard". That is, "/payInfo[amount>100]/creditCard" $\Rightarrow_2$ "/payInfo/amount" $\wedge$ "/payInfo/creditCard".*

Let *e"* denote an XPath expression returned after Step 2.

Step 3. Eliminate logical "|" and "and" operators in each XPath expression *e"* by rewriting the expression into several ones logically connected with "$\wedge$" or "$\vee$".

To match the notion of path in Definition 1, every XPath expression after Step 2 containing the logical operators "|" and "and" is substituted by a set of shorter XPath expressions, which are logically connected via "$\wedge$" or "$\vee$".

Example 6 *The XPath expression e'' = "/payInfo/(creditCard|cash)/name" can be viewed as two disjunctive expressions: e_1''' = "/payInfo/creditCard/name" and e_2''' = "/payInfo/cash/name", denoted as e'' $\Rightarrow_3$ e_1''' V e_2'''.*
Similarly, the expression "/payInfo/creditCard[name and dueDate]" can be equally transformed into "/payInfo/creditCard/name" Λ "/payInfo/creditCard/dueDate".

After undergoing the above three steps, an original XPath expression is transformed into a set of *simple XPath expressions*, each of which contains no ancestor-descendant relationship between every two consecutive nodes, no value constraints on nodes, and no logical operators ("|") and ("and"). A simple XPath expression obtained is now consistent with the definition of path in Section 3.

Example 7 *From an original XPath expression containing a predicate constraint and operator ("|") like "/payInfo[amount>100]/(creditCard|cash)/name", we derive three simple XPath expressions: "/payInfo/amount" Λ ("/payInfo/creditCard/name" V "/payInfo/cash/name").*

4.3 Identification of Candidate DTDs and Documents for a Query

On the basis of simple XPath expressions generated from XPath query expressions, we now define the concepts of candidate DTDs and documents for a given query. An XML DTD is called a *candidate DTD* for a query, if for every simple Xpath expression derived from the query, there *possibly* exists a path *p* in the DTD, that matches this simple XPath expression.

In a similar fashion, we define that an XML document is a *candidate document* for a query, if and only if: 1) its DTD is a candidate DTD;[1] and 2) it *possibly* satisfies all predicate constraints enforced on the nodes in the XPath query expression.

In the following, we start with the identification of candidate DTDs, followed by the identification of candidate documents for each candidate DTD that has been identified.

I. Identify Candidate DTDs by Hashing Paths

[1] Recall that we assume the availability of DTD for each document in this study.

Given a query, to check out which encrypted DTDs are candidate DTDs, for each simple XPath expression derived from the query, we match it to a path *p*, and compute the hash value for *p* using the same hash function *HashFunc*(*p*) (Algorithm 3.1) while encoding the DTDs. According to the hash value (i.e., bucket address) returned, we consult with the corresponding bucket in the hash table *DTDHashTable*$_{|p|}$, which gives all the identifiers of DTDs that may possibly contain path *p*. The rationale for this is straightforward: *if path p is present in the DTD, it will be hashed to the bucket in DTDHashTable*$_{|p|}$*, leaving a mark for this DTD in the bucket entry*.

Example 8 *Suppose a query consists of only one simple XPath expression, corresponding to the path p=(payInfo/creditCard/dueDate). Referring to the DTD encoding schema illustrated in Figure 4, where s=4 and SizeDTDHashTable*$_2$*=8, its hash value is computed as follows:*

Step 1: ChopName("payInfo", 4) = "payI", ChopName("creditCard", 4) = "cred", ChopName("dueDate", 4) = "dueD".

Step 2: Base26ValueOf("payI") = 264272, Base26ValueOf("cred") = 46751, Base26ValueOf("dueD") = 66355.

Step 3: HashFunc(payInfo/creditCard/dueDate)

$$= (\mathit{Base26ValueOf}(\text{“PayI”}) * 10^2 + \mathit{Base26ValueOf}(\text{“cred”}) * 10^1 + \mathit{Base26ValueOf}(\text{“dueD”}) * 10^0) \bmod \mathit{SizeDTDHashTable}_2$$

$$= (264272*100 + 46751*10 + 66355) \bmod 8 = 1$$

Due to its hash value 1, we can be sure that the example dtd$_2$ *does not contain that path, since the entry at address 1 in DTDHashTable*$_2$ *only signifies dtd*$_1$*. As a result, only dtd*$_1$ *will be returned as the candidate DTD, dtd*$_2$ *and its associated conforming documents can thus be discarded from further consideration.*

II. Identify Candidate Documents by Hashing Element/Attribute and Content/Value Pairs

After pre-selecting the candidate DTD set for the given query, we are now in the position to filter out candidate documents for each candidate DTD. At this stage, various value constraints in the form of [c_{name} θ c_{val}], (where c_{name} denotes the name of an element/attribute node, θ is one of the operators in $\{=, \neq, <, \leq, >, \geq\}$, and c_{val} denotes the element content/attribute value), on path nodes are taken into consideration. Clearly, a candidate document must not violate any of the value constraints specified within the XPath query expression. We perform such kind of examination based on the document encodings, i.e., $DOCHashTable_{dtdi}$ (where $dtd_i \in \Sigma_{DTD}$).

Taking the constraint [c_{name} θ c_{val}] for example, we first hash the node name c_{name} (i.e., a path containing only one node) into $DOCHashTable_{dtdi}$ via hash function $HashFunc(c_{name})$. Meanwhile, we also calculate the range identifier of c_{val} using the order preserving function $MapFunc(c_{val})$. Finally, we compare each entry value v linked to the bucket address $HashFunc(c_{name})$ in $DOCHashTable_{dtdi}$: if $\exists v$ (v θ $MapFunc(c_{val})$), then the constraint [c_{name} θ c_{val}] possibly holds.The associated document where v resides is then returned as the candidate document.

Example 9 *Assume a query embeds an XPath expression "/payInfo/creditCard [@limit>2000]/name", which enforces a constraint [@limit>2000] on creditCard element. Referring to the document encoding schema in Figure 5, where s=4 and* $SizeDOCHashTable_{dtd1}=4$. *We have HashFunc(limit) = 0 and MapFunc(2000) = 2. Since all the mapped values at address 0 in* $\$DOCHashTable_{dtd1}$ *are either 1 or 0, which is not greater than 2 (=MapFunc(2000)), therefore, the example document is not a candidate document for this query, and can thus be discarded.*

Correctness. The key to the correctness of the above candidate identification method lies in the following lemma.

Lemma 1 *The identified candidate DTDs (candidate documents) form the superset of the target DTDs (candidate documents). That is,* $C_{DTD} \subseteq T_{DTD}$ *and* $C_{DOC} \subseteq T_{DOC}$.

Proof 1 *Stage I only removes DTDs which surely do not belong to a query target. This is because if an XML DTD contains a path matching a simple XPath expression derived from the query, the corresponding hash entry will surely contain an indicator to this DTD, which thus returning the DTD as a candidate.*

Similarly, Stage II does not drop out potential target documents since any candidate document to be queried may possibly satisfy the value constraints, as specified in the XPath expressions.

Lemma 1 ensures the completeness of our candidate pre-selection phase.

4.4 Space and Time Complexity

The space requirement of the proposed strategy lies in the maintenance of two sets of hash tables, $DTDHashTable_l$ ($0 \leq l \leq max\_pathLen$) and $DOCHashTable_{dtdi}$ ($1 \leq i \leq num\_dtd$); That is, $Space = S_{DTDHashTables} + S_{DOCHashTables}$.

Since each entry of the DTD hash tables contains a list of DTD identifiers whose paths have been hashed into the bucket, the space complexity of the total DTD hash tables is thus $S_{DTDHashTables} = \sum_{l=0}^{max\_pathLen} SizeDTDHashTable_l * N_{collision\_dtd} * Size(dtdID)$, where $N_{collision\_dtd}$ is the average number of colliding DTDs that are hashed into the same bucket, and $Size(dtdID)$ is the size used to store a DTD identifier.

Similarly, the space complexity of the total DOC hash tables is $S_{DOCHashTables} = \sum_{i=1}^{num\_dtd} SizeDOCHashTable_{dtdi} * N_{collision\_doc} * (Size(docID) + Size(valID))$, where $N_{collision\_doc}$ is the average number of colliding documents that are hashed into the same bucket, $Size(docID)$ and $Size(valID)$ are the size of a document identifier and the size of a content/value identifier.

When a query is issued at run-time, the time overhead incurred includes the selection of candidate DTDs and candidate documents for each candidate DTD by computing the hash values of the paths and checking the content/value pairs. That is, $Time = T_{select\_candidate\_DTD} + T_{select\_candidate\_DOC}$.

Since the candidate DTD selection process just simply returns a linked list of DTD identifiers underneath the bucket of the resulting hash address, the computational cost is constant, i.e., $T_{select\_candidate\_DTD} = O(1)$.

The identification of candidate documents is also to compute the hash address of element tag/attribute name. In the case that there exists a query constraint like [*name* θ *value*], it will check the content/value identifier of each item linked underneath the bucket, and return the satisfying documents' identifiers. Therefore,

$T_{select\_candidate\_DOC} = O(N_{candidate\_DTD}*(1+N_{collision\_doc})) = O(N_{candidate\_DTD}*N_{collision\_doc})$,

where $N_{candidate\_DTD}$ is the number of candidate DTDs identified for the query.

In summary, the total time overhead of the proposed query strategy is:

$Time = T_{select\_candidate\_DTD} + T_{select\_candidate\_DOC}$.

$= O(1) + O(N_{candidate\_DTD}*N_{collision\_doc})$

$= O(N_{candidate\_DTD}*N_{collision\_doc})$.

5 Performance Study

To assess the performance of the proposed query strategy, we conducted a series of experiments on both synthetic and real-life data. The methods used to generate synthetic data and query workload consisting of a set of XPath expressions are described in Subsection 5.1 and 5.2, respectively. Subsection 5.3 describes two major performance measurements, namely *precision rate* and *mistake rate*. We report our experimental results obtained from synthetic data in Subsection 5.4 and from real data in Subsection 5.5.

5.1 Generation of Synthetic Data

The generation of synthetic data is divided into two steps. We first generate synthetic XML DTDs, and for each DTD, we then generate its conforming XML documents. Table 3 summarizes the parameters used and their settings. The number of encrypted DTDs in the database is *num_dtd*. Each DTD has *num_doc_per_dtd* encrypted conforming documents. In total, we have *num_dtd* * *num_doc_per_dtd* encrypted documents in the database.

Parameter	Meaning	Setting	Default
Database Parameters			
num_dtd	Number of encrypted DTDs in the database	100 – 500	200
num_doc_per_dtd	Number of encrypted documents per DTD	100 – 500	200
db_size	Database size equal to *num_dtd***num_doc_per_dtd*	10K – 250K	40K
XML Tree Structure (Topology) Parameters			

max_depth	maximum depth of trees, also maximum length of XPath query expressions	4 – 12	10
max_fanout	maximum number of children per node	3 – 7	5
node_name_size	length of node names (i.e., number of letters per node name)	4	4
letter_set	letter set from which node names are constructed	{a…f} – {a..z}	{a..f}
|*letter_set*|	cardinality of the letter set	{a..f}	6
XML Content Parameters			
max_frequency	maximum appearance of leaf nodes	1 – 9	5
num_distinct_content	number of distinct content values for leaf nodes	5000 - 10000	10000
Hash Table Parameters			
$SizeDTDHashTable_l$ ($0 \leq l \leq max\_depth$)	size of DTD hash table for paths of length *l*	10000 – 60000	55000
$SizeDOCHashTable_{dtdi}$ ($1 \leq i \leq num\_dtd$)	size of document hash table for dtd_i	4 - 14	8
Query Workload Parameters			
num_Xpath_per_dtd	number of XPaths derived from each DTD	3 – 8	3
num_query	number of queries (XPath expressions), equal to num_XPath_per_dtd * num_dtd	600 - 1600	600

Table 3. Parameters and settings

Step 1: *Generating XML DTDs.*

The core in generating every XML DTD for this study is to specify the topology of its tree structure, as depicted in Figure 2 (c). The maximum depth of the tree is restricted to *max_depth*. We use a normally distributed random variable with mean equal to *max_depth*/2 and variance equal to 1 to decide the depth of the tree. Each tree node has 0 to *max_fanout* child nodes, whose number is exponentially determined with mean equal to *max_fanout*/2. Leaf nodes have no children. The whole tree structure is generated in a depth-first manner. The first level of the tree contains only one root node.

After obtaining the tree structure, we assign string names to its nodes, which represent either XML element tags or attribute names. Since node names must be chopped into a fixed size *node_name_size*=4 in our method, we randomly select 4 letters from a letter set *letter_set* to construct one node name like "*abaf*", "*ecdq*", etc. By varying the cardinality of this letter set, we allow to generate duplicate names with different possibilities Basically, the smaller the cardinality, the higher chances to generate the same node names within and/or across different DTDs' tree structures. Several options for the letter set are utilized in our experiments, including {a...z}, {a...f}, {a...h}, {a...j}, {a...l} and {a...n}, with cardinality 26, 6, 8, 10, 12, 14, respectively.

Step 2: *Generating conforming XML documents for each DTD.*

We generate *num_doc_per_dtd* conforming documents for each DTD returned from Step 1. These documents differ in content values for the leaf nodes of the DTD tree structure. The content values are integers, representing the identifiers of element contents and/or attribute values. There are *num_distinct_content* different identifiers, which are uniformly selected in a round-robin order and assigned to the leaf nodes. We allow one element node to appear several times, but with possibly different content values within one XML document. The element frequency is randomly determined with a maximum value *max_frequency*.

5.2 Generation of Query Workload

We simulate query workload by generating a set of valid XPath expressions with no duplicates. Taking each DTD as input, we derive *num_XPath_per_dtd* XPath expressions that conform to this DTD, which resulting in *num_XPath_per_dtd* * *num_dtd* XPath expressions in total. Similar to the DTD generation, before generating an XPath expression, we randomly pick a number from 1 to *max_depth* as its path length limitation. The actual length of the XPath expression could be less than, but definitely should not exceed this value. From the first level downwards, we are at 80% of chance to include a node at each level in the XPath. A non-selected level implies a ancestor-descendant ("//") relationship in the XPath. For the selected levels, we randomly choose nodes, one node per level, and then assign them to the XPath expressions. Nodes selected at the lower levels must be the descendants of the nodes selected at the upper levels. For nodes coming from two consecutive levels, there will be a parent-child ("/") relationship between them in the XPath. The constraint in the form of "[*name* θ *valId*]" is finally assigned to each XPath expression. For the sake of document filtering testing, here, both *name* and *valId* are taken from the DTD and one of its conforming documents. Currently, we do not consider the wildcard operator ("*"), and θ is limited to "=" operator during the generation of XPath expressions.

Note that although each XPath expression is derived from a single DTD, it may conform to several DTDs' tree structures due to multiple generation possibilities of the same node names for different DTDs, as described in Subsection 5.1.

5.3 Performance Measures

Throughout the experiments, the following two measures are used to evaluate the performance of candidate DTD and document filtering.

Precision rate - Among the pre-selected candidate DTDs (documents) for a query, how many of them are the real query target DTDs (documents).

$Precision_{DTD}$ = *num of target DTDs / num. of candidate DTDs* $= |T_{DTD}| / |C_{DTD}|$

$Precision_{DOC}$ = *num of target DOCs of a DTD / num. of candidate DOCs of a DTD*

$= |T_{DOC}^{d}| / |C_{DOC}^{d}|$

where C_{DOC}^{d} and T_{DOC}^{d} represent the set of candidate documents and target documents, and these documents conform to the same candidate DTD.

Mistake rate - Among the whole set of DTDs (documents) in the database, how many of them are wrongly pre-selected as candidates, while they are in fact not the query targets.

$Mistake_{DTD}$ = *(num of candidate DTDs – num of target DTDs) /*

(num of DTDs in the database – num of target DTDs)

$= (|C_{DTD}| - |T_{DTD}|) / (|\Sigma_{DTD}| - |T_{DTD}|)$

$Mistake_{DOC}$ = *(num of candidate DOCs of a DTD – num of target DOCs of a DTD) /*

(total num of DOCs of a DTD – num of target DOCs of a DTD)

$= (|C_{DOC}^{d}| - |T_{DOC}^{d}|) / (|\Sigma_{DOC}^{d}| - |T_{DOC}^{d}|)$

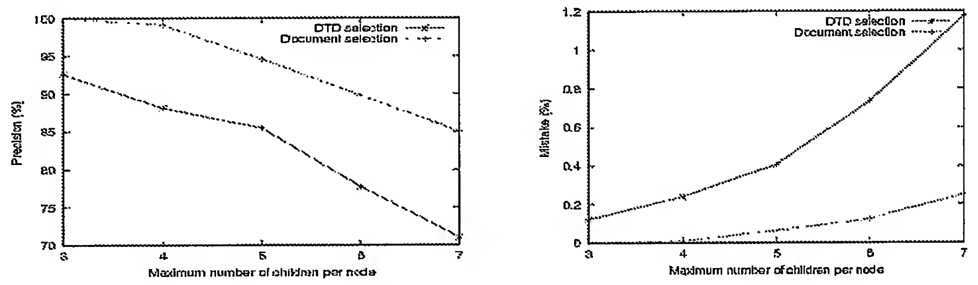
where Σ_{DOC}^{d} represents the whole set of documents that conform to a candidate DTD.

5.4 Experiments on Synthetic Data

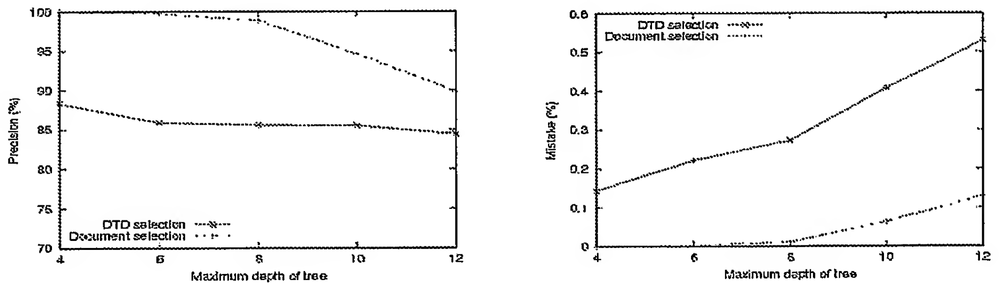
The synthetic data sets enable us to study the scalability of the proposed search strategy over encrypted XML documents under arbitrary complexity, controlled by tuning a set of parameters. We have conducted five sets of experiments on a Sun UltraSPARC-IIi with a 440 MHz CPU clock rate and 256 MB main memory.

5.4.1 Effect of XML Tree Structures

The first set of experiments investigates how the topology of XML trees affect the candidate DTD and document filtering for a given query, when two characteristic parameters (i.e., *max_fanout* and *max_depth*) of a tree are varied. Unless explicitly specified, in the following, the rest of the parameters are set to their default values given in Table 3. The experimental results reported here are the average performance under the whole query workload.



(a) Maximum fanout of XML trees vs. filtering precision and mistake rates



(b) Maximum depth of XML trees vs. filtering precision and mistake rates

Figure 6. Performance under different XML tree structures

When we increase the maximum number of children per node, the precision rates of both DTD filtering and document filtering get decreased. This is because the obtained XML trees become more bushy, leading to more paths in DTD trees and also more content value pairs in documents being hashed. Therefore, the chance of hash collision gets increased, resulting in more wrongly selected candidate DTDs and candidate documents, which are not the real query targets. Note that a degraded precision rate is always accompanied with an increased mistake rate. For example, in Figure 6 (a), the precision rate of candidate DTD filtering decreases from 88.1% at *max_fanout*=4 to 77.7% at *max_fanout*=6, while its mistake rate increases from 0.24% to 0.73%.

Similarly, when we increase the allowable depth of XML trees, the number of paths embedded within the DTDs, as well as the number of content values for more leaf nodes within the conforming documents, gets increased, leading to a higher chance for wrongly candidate identification according to the document hash tables, as demonstrated in Figure 6 (b).

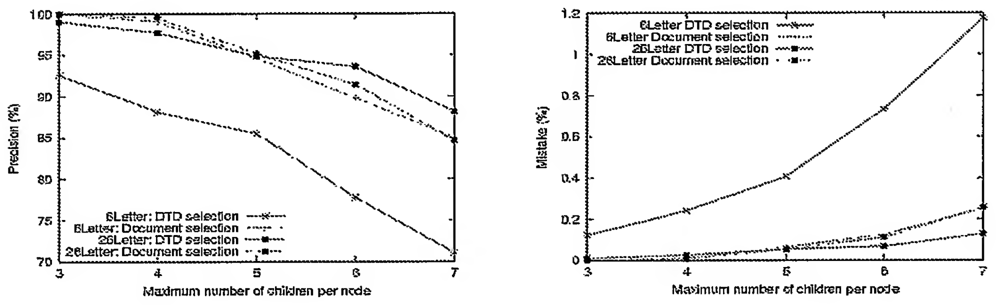
5.4.2 Effect of Tree Overlap Degrees

The second set of experiments is designed to study the candidate selection performance when there exist certain overlaps across different XML tree structures. That is, some nodes and their constructed paths may appear in several XML trees. By tuning the *letter_set* parameter which is used to assign string names to nodes, we achieve different levels of node/path overlaps.

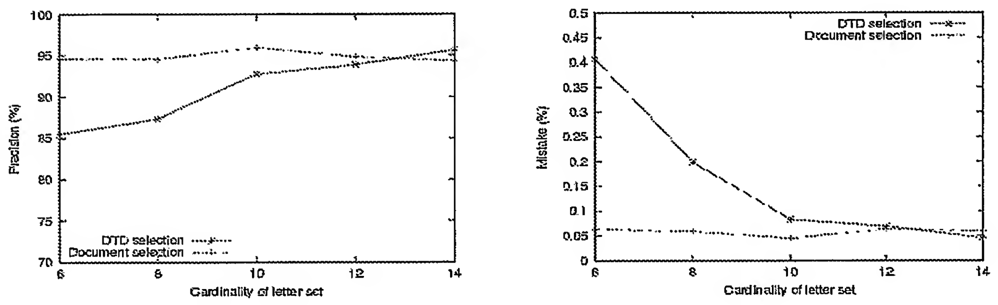
Figure 7 (a) presents two groups of results, where one letter set is {a, b, ..., f} containing 6 letters, and the other is {a, b, ..., z} containing 26 letters. An important observation made from it is that the DTD selection behavior under |*letter_set*|=6 is always far superior over the one under |*letter_set*|=26. For example, at *max_fanout*=5, the precision rate of 26-Letter candidate DTD selection is 94.8%, while that of 6-Letter candidate DTD selection is 85.5%. The former is (94.8%-85.5%)/85.5% ≈ 10.9% more than the latter. This is due to the fact that a larger letter set inevitably produces lower overlaps across different trees. Confronted with more distinct node names and thus paths, the hash function on paths can apparently perform better in real candidate identification.

However, compared to the candidate DTD selection, the influence of letter sets on candidate document selection is not so much. For example, the precision rate of 26-Letter candidate document selection at *max_fanout*=5 is 95.2%, which is slightly higher than the precision rate of 6-Letter document selection 94.6%. The reason for this is that we identify candidate documents by checking element tags/attribute names, as well as their content/value identifiers. The node names, despite of the existence of duplication, contribute partly to the candidate identification.

We further test different letter sets with various cardinalities spanning from 6 to 12. The results obtained in Figure 7 (b) coincide with our expectation.



(a) Comparison between 26-letter and 6-letter sets with different *max_fanout*



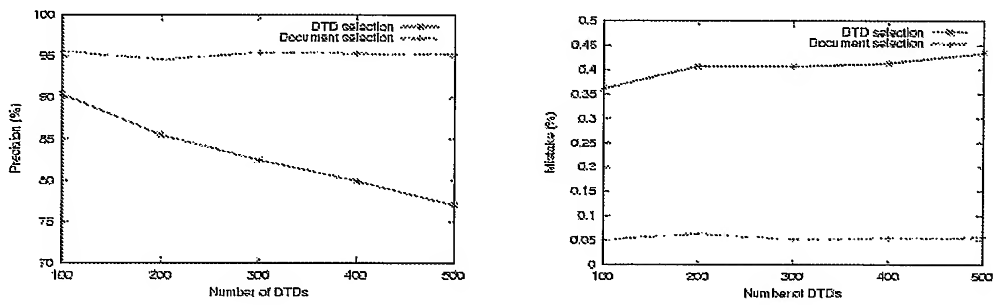
(b) Cardinality of letter sets vs. filtering precision and mistake rates

Figure 7. Effect of tree overlap degrees

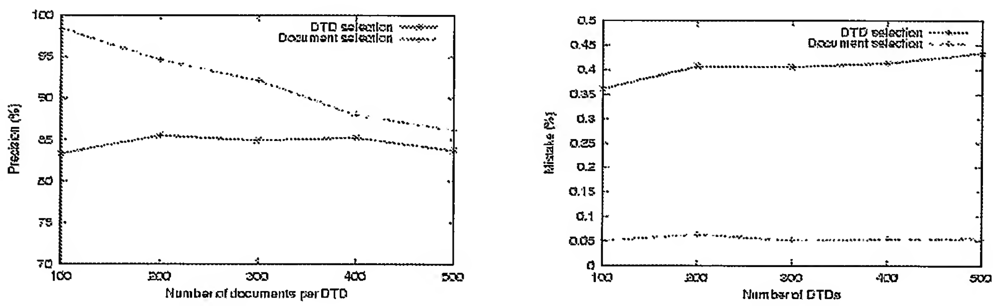
5.4.3 Effect of Database Sizes

Our third experiment examines the scale-down behavior of candidate DTD and document filtering, while we enlarge the total number of DTDs in the database, *num_dtd,* from 100 to 500 and the total number of documents per DTD, *num_doc_per_dtd,* from 100 to 500.

As shown in Figure 8 (a), the increase of total DTD number in the database hardly affects the candidate document filtering. This is not surprising, since each time the document selection scope is limited to one DTD's conforming document set rather than multiple document sets from different DTDs. On the contrary, the increase of total document number per DTD hardly changes the behavior of candidate DTD selection as shown in Figure 8 (b), since it identifies the candidate DTDs according to the path-based hash tables *DTDHashTable*$_l$ (where $1 \leq l \leq max\_depth$), and not for element/attribute content pairs in the documents. Both candidate DTD filtering and candidate document filtering scale linearly downward with the growing of total DTD number in the database and total document number per DTD respectively, as illustrated in Figure 8.



(a) Number of DTD vs. filtering precision and mistake rates



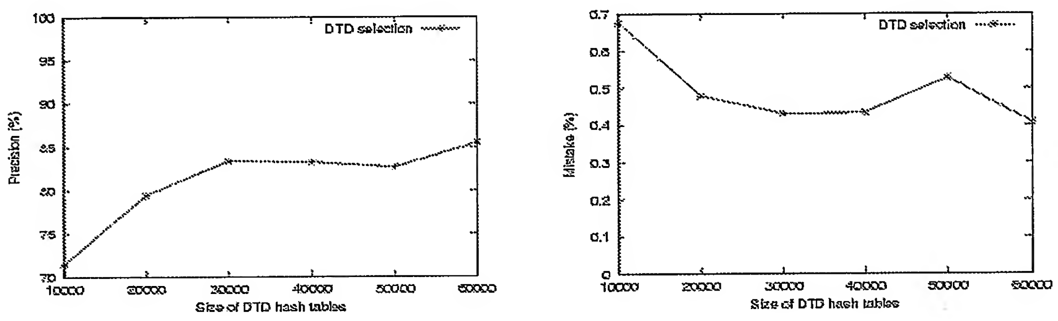
(b) Number of documents per DTD vs. filtering precision and mistake rates

Figure 8. Results of scale down experiments

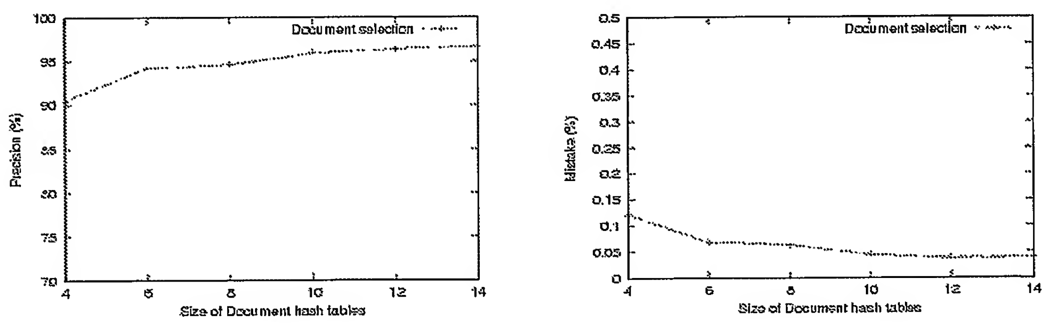
5.4.4 Effect of Hash Table Sizes

Observing that hash table sizes have a direct impact on hash collision, we examine the filtering performance under various hash table sizes. All the DTD hash tables for paths of different lengths are set to the same size, and also all the DOC hash tables for different DTDs have the same size. The experimental results are presented in Figure 9.

Since the filtering of candidate DTDs and documents works on two different kinds of hash tables, namely, *DTDHashTable*$_l$ (where $1 \leq l \leq max\_depth$) and *DOCHashTable*$_{dtdi}$ ($1 \leq i \leq num\_dtd$), changing the size of DTD hash tables (document hash tables) may only affect the behavior of DTD filtering (document filtering) and not both. As both Figure 9 (a) and (b) show, with more hash entries, the problem of hash collision on paths (content value pairs) can be alleviated. The performance exhibits an increasing tendency.



(a) DTD hash table size vs. filtering precision and mistake rates



(b) DOC hash table size vs. filtering precision and mistake rates

Figure 9. Performance under different hash table sizes

5.4.5 Effect of Query Workload

Figure 10 shows the candidate DTD and document filtering precision and mistake rates when we vary the total number of XPath queries *max_query* from 600 to 1600. It's interesting to note that given a database of a fixed size, the candidate selection strategy are rather robust under different query workload.

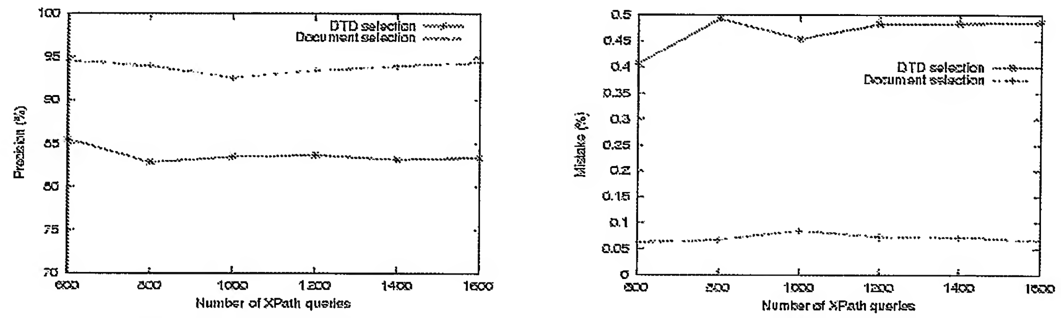


Figure 10. XPath query workload versus filtering precision and mistake rates

5.5 Data *Experiments on Real Life Data*

To study the applicability of the proposed search strategy, we conduct experiments on real-life XML data, obtained from 3 major sources, i.e., ACM SIGMOD Record [2], Shakespeare's Plays marked up by Jon Bosak [3], and XML.org which provides industrial submitted DTDs [4], as shown in Table 4. The total number of DTDs used for testing the candidate DTD selection is 20. The maximum depth of the DTDs is 8, and the maximum fanout (i.e., the maximum number of children of tree nodes) is 23.

DTD Set	Document Set

[2] http://www.acm.org/sigmod/record/xml
[3] http://www.oasis-open.org/cover/bosakShakespeare200.html
[4] http://www.xml.org/xml/focus_areas.shtml

SIGMOD	*"ProceedingsPage.dtd"*	16 documents
	"OrdinaryIssuePage.dtd"	30 documents
	"SigmodRecord.dtd"	1 document
	"HomePage.dtd"	Unavailable
Shakespeare	*"play.dtd"*	37 documents
XML.org	*15 industrial dtds*	Unavailable

Table 4. XML real-life data set

Considering the very long lengths (around 40 letters) of element tags and attribute names used in these real-life DTDs, to convert a string node name into an integer, instead of using *Base26ValueOf* function, we adopt the function *SumValueOf* $(x_{n\,i,1}\; x_{n\,i,2} \ldots x_{n\,i,s}) = \mathit{offset}(x_{n\,i,1}) + \mathit{offset}(x_{n\,i,2}) + \ldots + \mathit{offset}(x_{n\,i,s}) = V_{n\,i}$. The hash function on path $(n_1/n_2/ \ldots /n_k)$ is adapted to: $\mathit{HashFunc}(n_1/n_2/ \ldots /n_k) = (V_{n\,1} * 2^{k-1} + V_{n\,2} * 2^{k-2} + \ldots + V_{n\,k} * 2^0) \bmod \mathit{SizeDTDHashTable}_{k-1}$.

We test two XPath query expressions. The first one (*OrdinaryIssuePage*[*number="3" and volume="23"*]) contains only the parent-child relationship between *OrdinaryIssuePage* and *number/volume*. The second one (*ProceedingsPage//authors*[*author="Hongjun Lu"*]) incorporates an ancestor-descendant relationship between *ProceedingsPage* and *authors*.

To filter out candidate DTDs from among the 20 DTDs for the queries, we first match XPath expressions to paths, as described in Subsection 4.2. The first query expression is matched to two conjunctive paths: "*OrdinaryIssuePage/number*" and "*OrdinaryIssuePage/volume*"; while the second one to another two conjunctive paths: "*ProceedingsPage*" and "*authors/author*". Here, we leave the content constraints out while doing candidate DTD selection. Table 5 presents the results while different DTD hash table sizes are used. As illustrated, by choosing an appropriate hash table size, we can even achieve 100% precision rate and 0% mistake rate.

Query 1: *OrdinaryIssuePage* [*number="3" and volume="23"*]				
SizeDTDHashTable	100	120	150	160
Precision (%)	50.0%	50.0%	50.0%	100.0%
Mistake (%)	5.26%	5.26%	5.26%	0.0%

Query 2: *ProceedingsPage//authors* [*author="Hongjun Lu"*]				
SizeDTDHashTable	100	120	150	160
Precision (%)	33.3%	100.0%	100.0%	100.0%
Mistake (%)	10.53%	0.0%	0.0%	0.0%

Table 5. Results of candidate DTD selection from real-life data

After filtering out candidate DTD for the given queries, the scope for candidate document selection is narrowed to the conforming document sets under the selected DTDs. We identify candidate documents by examining the document hash tables established for these DTDs. Recall that each bucket of a document hash table records a list of content values, together with the identifiers of documents in which they stay. For instance, the document hash table for *ProceedingsPage.dtd* accommodates the content/value identifiers for the elements/attributes including *volume, number, month, year, confYear, conference, date, location, articleCode, authorPosition, size, inline, href, xml:link, addidtionalMaterial, initPage* and *endPage.*

To locate candidate documents for Query 1 which has two value constraints (*volume*="23" and *number*="3"), we first compute the hash addresses of *volume* and *number*, and then go through the value lists underneath the corresponding hash buckets to compare their value identifiers against 23 for *volume* and 3 for *number*, from which we then select the documents having the same *volume* value and *number* value. For both queries, the performance of our candidate document filtering reach 100% precision rate and 0% mistake rate.

6 Conclusion

In this paper, we employ the efficient hash technique to compute encodings associated with each encrypted XML data to allow effective pre-filtering of candidate data for a given query, expressed in terms of XPath expressions. A three-staged framework, consisting of *query preparation, query pre-processing* and *query execution*, for efficient querying of encrypted XML data is outlined. The time and space complexity of the proposed strategy is analyzed.

We overview our experience using the method on both synthetic and real-life data, and have made several interesting observations from the experimental results.1) The size of hash tables is an important factor in the pre-selection of candidate DTDs and documents for a query. By careful tuning the sizes of DTD and DOC hash tables, we can achieve good precision rate and mistake rate, as shown throughout the experiments. 2) The behaviors of candidate DTD filtering and candidate document filtering scale down linearly with the increase of total DTD number in the database and document number per DTD. 3) The topologies of XML data trees, including the maximum depth and fanout of the trees, affect the candidate filtering performance. A complex tree can cause more hash collisions, thus inevitably degrading the hash-based filtering results. 4) Also, for the same reason, the more overlaps between different DTDs (documents), the worse the candidate filtering performance.

This paper makes a first step towards querying over encrypted XML data, with a number of interesting problems remaining for future work. First, the setting of hash table sizes is subject to both XML data sources and query workload. It is interesting to investigate this issue from a theoretical point of view. Second, tests with more complex query conditions, including wildcard operator ("*"), position constraints, etc., are necessary in order to provide a more comprehensive picture over the filtering performance. Third, some enhancements can be made to further improve the query performance. For instance, in this study, we hash paths which only include the parent-child ("/") relationship between every two successive nodes. It would also be interesting to compute hash values for paths taking into account the ancestor-descendant relationship ("//"). Furthermore, supporting efficient blind queries when both XML data to be queried and XML query expressions themselves are encrypted is another important issue worth exploration.

References

[1] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proc. the 26th Intl. Conf. on Very Large Data Bases*, pages 53-64, Carior, Egypt, September 2000.

[2] C. Chan, P. Felber, M. Carofalakis, and R. Rastogi. Efficient filtering of XML documents with Xpath expressions. In *Proc. the Intl. Conf. on Data Engineering*, California, USA, February 2002.

[3] World Wide Web Consortium. Extensible markup language (XML) 1.0. http://www.w3.org/TR/REC-xml, October 2000.

[4] World Wide Web Consortium. XML encryption requirements. http://www.w3.org/TR/xml-encryption-req, March 2002.

[5] World Wide Web Consortium. XML encryption syntax and processing. http://www.w3.org/xmlenc-core/, August 2002.

[6] World Wide Web Consortium. XML path language (XPath) 2.0. http://www.w3.org/TR/xpath20/, November 2002.

[7] World Wide Web Consortium. Xquery 1.0: an XML query language. http://www.w3.org/TR/xquery/, November 2002.

[8] H. Hacigümüş, B. Lyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proc. the ACM SIGMOD Intl. Conf. on Management of Data*, pages 216-227, Wisconsin, USA, June 2002.

[9] H. Hacigümüş, B. Lyer, and S. Mehrotra. Providing database as a service. In *Proc. the Intl. Conf. on Data Engineering*, California, USA, February 2002.

[10] W. Jonker. XML and secure data management in an ambient world. In *Intl. J. Computer Systems Science & Engineering*, 2002 (to appear).

[11] D. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proc. the IEEE Symposium on Security and Privacy*, 2000.

While the invention has been described in connection with preferred embodiments, it will be understood that modifications thereof within the principles outlined above will be evident to those skilled in the art, and thus the invention is not limited to the preferred embodiments but is intended to encompass such modifications. The invention resides in each and every novel characteristic feature and each and every combination of characteristic features. Reference numerals in the claims do not limit their protective scope. Use of the verb "to comprise" and its conjugations does not exclude the presence of elements other than those stated in the claims. Use of the article "a" or "an" preceding an element does not exclude the presence of a plurality of such elements.

'Computer program' is to be understood to mean any software product stored on a computer-readable medium, such as a floppy disk, downloadable via a network, such as the Internet, or marketable in any other manner.

CLAIMS:

1. A method of searching in a collection of documents, the documents having a tree-like structure and each document in the collection of documents complying with at least one document structure definition in a collection of document structure definitions, comprising the steps of:

receiving a certain branch;

determining a subset of the collection of document structure definitions, each document structure definition in the subset allowing the certain branch to exist in a document complying to the document structure definition;

determining a subset of the collection of documents, the subset of documents comprising all documents of the collection of documents complying to any one of the document structure definitions in the subset; and

searching for at least part of the certain branch in each document.

2. A method as claimed in claim 1, wherein a further step comprises attempting to decrypt each encrypted document in the subset of documents.

3. A method as claimed in claim 1, wherein the step of determining a subset of the collection of documents comprises calculating a number for at least part of the certain branch by applying a hash function to the at least part of the certain branch and looking up which documents are mapped to the calculated number in a mapping from number to documents, the mapping being associated with a document structure definition of the subset of document structure definitions and the documents in the mapping complying to the document structure definition.

4. A method as claimed in claim 3, wherein a further step comprises receiving a certain value associated with the certain branch, the mapping further comprises an association between a document in the mapping and a value domain partition, and the step of determining a subset of the collection of documents further comprises checking whether a value domain partition associated to a document mapped to the calculated number matches a

further value domain partition, the further value domain partition comprising the received value.

5. A method as claimed 1, wherein the step of determining a subset of the collection of documents comprises looking up, in a mapping from document structure definition to documents, which documents comply to any one of the subset of document structure definitions.

6. A method as claimed in claim 1, wherein the step of determining a subset of the collection of document structure definitions comprises calculating a further number for at least part of the certain branch by applying a further hash function to the at least part of the certain branch and looking up which document structure definitions are mapped to the calculated number in a mapping from number to document structure definitions.

7. A method as claimed in claim 1, wherein the step of determining a subset of the collection of document structure definitions comprises attempting to decrypt each encrypted document structure definition in the collection of document structure definitions and attempting to determine for each document structure definition whether the document structure definition allows the certain branch to exist in a document complying to the document structure definition

8. A computer program product enabling a programmable device to carry out a method as claimed in claim 1.

9. A method of indexing a collection of documents, the documents having a tree-like structure and each document in the collection of documents complying to at least one document structure definition in a collection of document structure definitions, comprising the steps of:

creating an empty index for each document structure definition of the collection of document structure definitions, the index mapping an integer from a range of integers to a document of the collection of documents;

calculating a number for at least a part of a branch in a document of the collection of documents by applying a hash function to the at least part of the branch, the number being limited to the range of integers and the calculation possibly producing a same number for different branches; and creating an entry in an index for a document structure definition to which said document complies, the entry comprising a mapping from said calculated number to said document comprising the branch.

10. A method as claimed in claim 9, wherein creating an entry in the index comprises associating the document in the mapping to a value domain partition, the value domain partition comprising a value associated with the branch.

11. A method as claimed in claim 9, wherein:

a further step comprises creating an empty further index in which each integer from a further range of integers can be mapped to a document structure definition;

a further step comprises calculating a further number for at least part of said branch by applying a further hash function to said branch, the further number being limited to the further range of integers and the calculation possibly producing a same further number for different branches, and

a further step comprises creating an entry in the further index, the entry in the further index comprising a mapping from the calculated further number to said document structure definition to which said document complies.

12. A computer program product enabling a programmable device to carry out a method as claimed in claim 9.

ABSTRACT:

The invention relates to a method of searching in a collection of documents, the documents having a tree-like structure and each document in the collection of documents complying with at least one document structure definition in a collection of document structure definitions. The method comprises the steps of:

receiving a certain branch;

determining a subset of the collection of document structure definitions, each document structure definition in the subset allowing the certain branch to exist in a document complying to the document structure definition;

determining a subset of the collection of documents, the subset of documents comprising all documents of the collection of documents complying to any one of the document structure definitions in the subset; and

searching for at least part of the certain branch in each document.

PCT/IB2004/051244